C212/A592 Lab 6

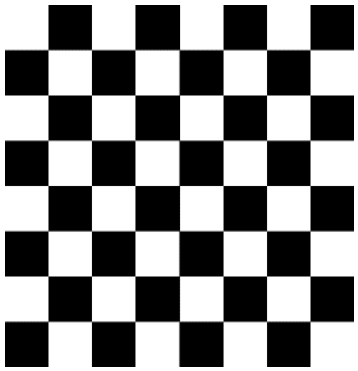
Intro to Software Systems

Instructions:

- Review the requirements given below and complete your work. Please compress all files (including main to test your work) into a zip file and submit it through Canvas.
- **Grading Scheme**: Part 1 (25 pts), Part 2 (100 pts): Total 125 points which includes 10 points for coding style.
- No Junit test case needed for this lab. But you should run methods in main(String[] args) as demos. Failure to do so will lose maximum of 15 points. See detail in canvas rubric

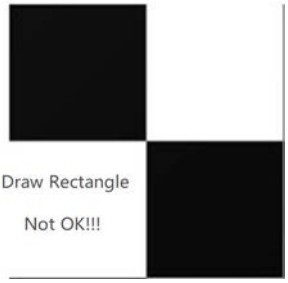
Part 1: CheckBoard of Triangles

Write a program with function **printCheckBoard(int size)** which draws the checkboard (given on the right). Use any combination of two colors.



The requirements are as follow:

1. For each block, please use triangles instead of squares



2. The input size determines the length and width of each small square in checkboard.
3. Number of blocks in each row should always be 8 (4 with color-1 and 4 with color-2).
4. Do not hardcode the board!! You should use loops to generate the board.
5. Use of scanner is not required. But you should print at least two checkboards in your main(String[] args) function with different **size.**

Part 2: The Game of Life

The Game of Life is a well-known mathematical game that gives rise to amazingly complex behavior, although it can be specified by a few simple rules. (It is not actually a game in the traditional sense, with players competing for a win.) Here are the rules. The game is played on a rectangular board. Each square can be either empty or occupied. At the beginning, you can specify empty and occupied cells in some way; then the game runs automatically. In each generation, the next generation is computed. A new cell is born on an empty square if it is surrounded by exactly three occupied neighbor cells. A cell dies of overcrowding if it is surrounded by four or more neighbors, and it dies of loneliness if it is surrounded by zero or one neighbor. A neighbor is an occupant of an adjacent square to the left, right, top, or bottom or in a diagonal direction. Figure 21 shows a cell and its neighbor cells.

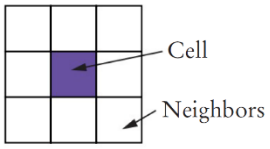


Figure 21
Neighborhood of a Cell

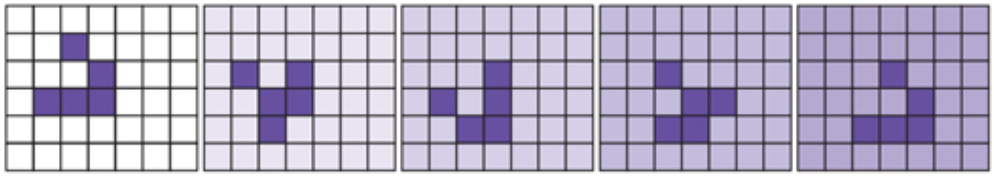
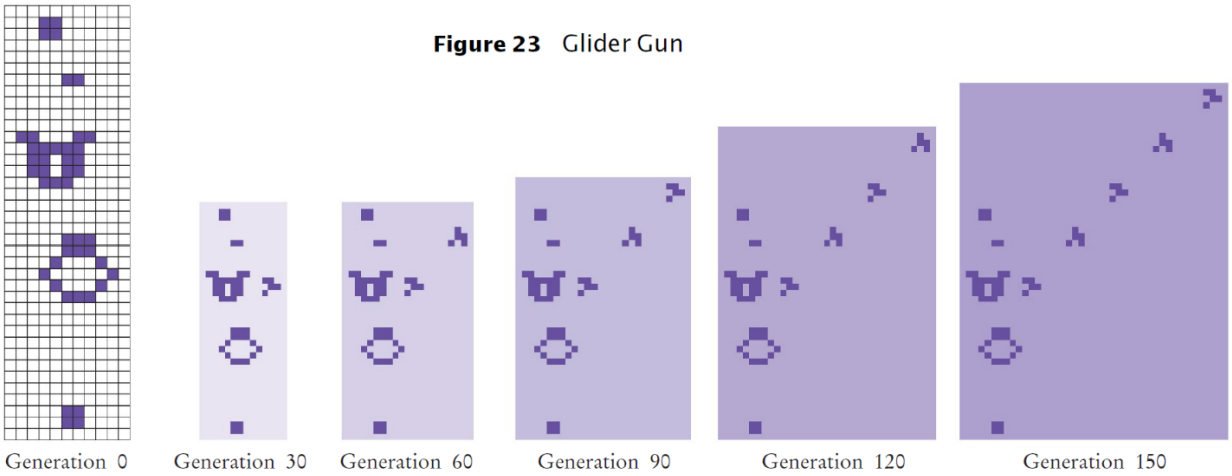


Figure 22 Glider

Many configurations show interesting behavior when subjected to these rules. Figure 22 shows a glider, observed over five generations. After four generations, it is transformed into the identical shape, but located one square to the right and below.

One of the more amazing configurations is the glider gun: a complex collection of cells that, after 30 moves, turns back into itself and a glider (see Figure 23).

Figure 23 Glider Gun



Write a class called **GameofLife** which perform the game of life with given pattern. The skeleton class is given as follow:

```
public class GameofLife {
        private boolean[][] board;
        private final int NUMROW;
        private final int NUMCOL;

        public GameofLife (boolean[][] initialBoard) { }
        public String toString() { }

        public void grow(int generation) { }
        private int checkNeighbor(int x, int y){ }
}
```

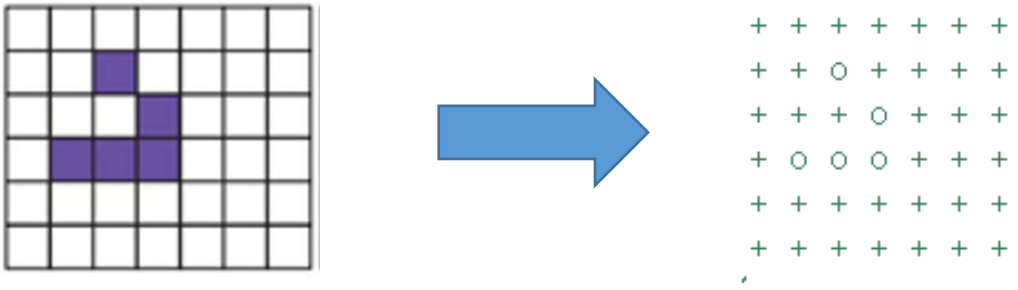
Instructions:

1. private Boolean[][] board:
   The **board** is the game board
   - occupied blocks are represented with true
   - empty blocks are represented with false.

2. public GameofLife(Boolean[][] initialBoard){ }
   In the constructor, you should:
   - Initiate **NUMROW** and **NUMCOL**.
   - Use for loops to deep copy **initialBoard** to **board**. Do not directly assign **initialBoard** to **board**

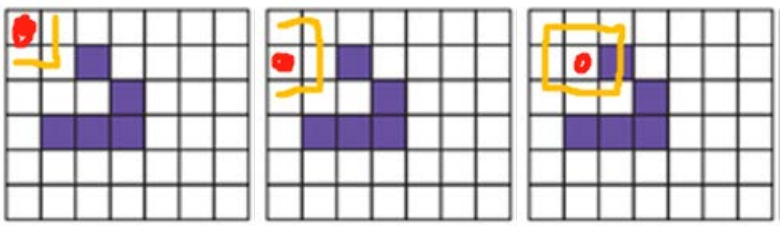
3. public String toString(){ }
   This returns a string that represents the board state.
   - For each occupied block, you should represent it as the lower-case letter “o”.
   - For each empty block, you should represent it as “+”.
   - For each block in a row, separate them with an empty space “ ”.

   For example:



4. public void grow(int generation, int freqPrint) { }
   This function modifies the **board**. It should also frequently print the entire board using **toString()** to show how much the board has changed.
   - The input **generation** controls how many generations your board is going to run.
   - The input **freqPrint** controls how frequently the board got printed. If freqPrint = 5, then the board is printed at every 5 generations. If freqPrint = 1, then the board is printed in each generation
   - In each generation, the program should start from the top left most block, check & update all blocks.
     - When checking, please use the helper function **checkNeighbor(x, y)** to get the number of occupied blocks in (x, y)'s neighbors as **M** (you should have better naming in your code)
     - if **M** is 3, set the block (x, y) to occupied.
     - If **M** is 2 and (x, y) itself is occupied, the block (x, y) keeps occupied
     - If **M** is 1 or 0 (loneliness), or greater than 3 (overcrowding), set the block to empty.
   - Time complexity is not considered. There could be certain algorithms that can speed up the game. Implementing such algorithm is not required.

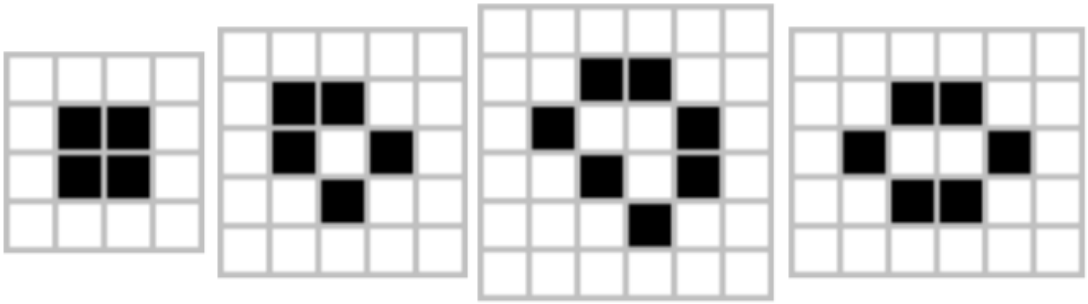
5. private int checkNeighbor(int x, int y){ }
   The function returns the number of occupied blocks of the neighbors of the block (x, y)
   - Neighbors are consisted by the blocks around the given block (x, y)
   - Neighbors does not include (x, y)
   - The maximum number of neighbors (x, y) could have is 8. If the given block (x, y) are on the edge, the neighbors is consisted by the possible surrounding block.
     For example,
     The neighbor of (0, 0) is (0, 1), (1, 0), (1, 1).
     The neighbor of (1, 0) is (0, 0), (0, 1), (1, 1), (2, 1), (2, 0).
     The neighbor of (1, 1) is (0, 0), (0, 1), (0, 2), (1, 2), (2, 2), (2, 1), (2, 0), (1, 0).



Testing:

To test your class, you should select at least one shape from each category listed as following. For your selected shape, you should generate the Boolean[][] board and create distinct GameofLife object using the board generated (in your main methods). And then test your grow(generation, freqPrint) method with generation = 100 and freqPrint = 20

- **Still Life**: the following patterns stay unchanged no matter how many generations



- **Oscillators**: The following patterns repeat itself within a certain period. Try to find the period if you can